# Dubbo 优雅停机演进之路

作者：9526xu

原文链接：https://ld246.com/article/1572772063674

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

一、前言

在 『ShutdownHook- Java 优雅停机解决方案』 一文中我们聊到了 Java 实现优雅停机原理。接下我们就跟根据上面知识点，深入 Dubbo 内部，去了解一下 Dubbo 如何实现优雅停机。

# 二、Dubbo 优雅停机待解决的问题

为了实现优雅停机，Dubbo 需要解决一些问题：

1. 新的请求不能再发往正在停机的 Dubbo 服务提供者。
2. 若关闭服务提供者，已经接收到服务请求，需要处理完毕才能下线服务。
3. 若关闭服务消费者，已经发出的服务请求，需要等待响应返回。

解决以上三个问题，才能使停机对业务影响降低到最低，做到优雅停机。

# 三、2.5.X

Dubbo 优雅停机在 2.5.X 版本实现比较完整，这个版本的实现相对简单，比较容易理解。所以我们以 Dubbo 2.5.X 版本源码为基础，先来看一下 Dubbo 如何实现优雅停机。

## 3.1、优雅停机总体实现方案

优雅停机入口类位于 AbstractConfig 静态代码中，源码如下：

```
static {
    Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
        public void run() {
            if (logger.isInfoEnabled()) {
                logger.info("Run shutdown hook now.");
            }
            ProtocolConfig.destroyAll();
        }
    }, "DubboShutdownHook"));
}
```

这里将会注册一个 ShutdownHook ，一旦应用停机将会触发调用 ProtocolConfig.destroyAll()。

ProtocolConfig.destroyAll() 源码如下：

```
public static void destroyAll() {
    // 防止并发调用
    if (!destroyed.compareAndSet(false, true)) {
        return;
    }
    // 先注销注册中心
    AbstractRegistryFactory.destroyAll();

    // Wait for registry notification
    try {
        Thread.sleep(ConfigUtils.getServerShutdownTimeout());
    } catch (InterruptedException e) {
```

```
        logger.warn("Interrupted unexpectedly when waiting for registry notification during shu
down process!");
    }

    ExtensionLoader<Protocol> loader = ExtensionLoader.getExtensionLoader(Protocol.class);
    // 再注销 Protocol
    for (String protocolName : loader.getLoadedExtensions()) {
        try {
            Protocol protocol = loader.getLoadedExtension(protocolName);
            if (protocol != null) {
                protocol.destroy();
            }
        } catch (Throwable t) {
            logger.warn(t.getMessage(), t);
        }
    }
    }
```

从上面可以看到，Dubbo 优雅停机主要分为两步：

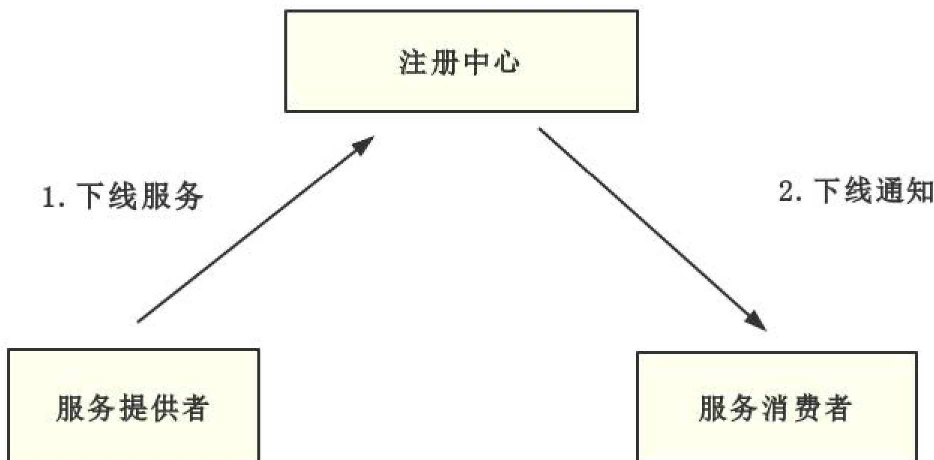
1. 注销注册中心
2. 注销所有 Protocol

## 3.2、注销注册中心

注销注册中心源码如下：

```
public static void destroyAll() {
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Close all registries " + getRegistries());
    }
    // Lock up the registry shutdown process
    LOCK.lock();
    try {
        for (Registry registry : getRegistries()) {
            try {
                registry.destroy();
            } catch (Throwable e) {
                LOGGER.error(e.getMessage(), e);
            }
        }
        REGISTRIES.clear();
    } finally {
        // Release the lock
        LOCK.unlock();
    }
}
```

这个方法将会将会注销内部生成注册中心服务。注销注册中心内部逻辑比较简单，这里就不再深入源，直接用图片展示。




ps: 源码位于：AbstractRegistry

以 ZK 为例，Dubbo 将会删除其对应服务节点，然后取消订阅。由于 ZK 节点信息变更，ZK 服务端会通知 dubbo 消费者下线该服务节点，最后再关闭服务与 ZK 连接。

通过注册中心，Dubbo 可以及时通知消费者下线服务，新的请求也不再发往下线的节点，也就解决面提到的第一个问题：新的请求不能再发往正在停机的 Dubbo 服务提供者。

但是这里还是存在一些弊端，由于网络的隔离，ZK 服务端与 Dubbo 连接可能存在一定延迟，ZK 通可能不能在第一时间通知消费端。考虑到这种情况，在注销注册中心之后，加入等待进制，代码如下：

```
// Wait for registry notification
try {
    Thread.sleep(ConfigUtils.getServerShutdownTimeout());
} catch (InterruptedException e) {
    logger.warn("Interrupted unexpectedly when waiting for registry notification during shutd
wn process!");
}
```

默认等待时间为 **10000ms**，可以通过设置 dubbo.service.shutdown.wait 覆盖默认参数。10s 只是个经验值，可以根据实际情设置。不过这个等待时间设置比较讲究，不能设置成太短，太短将会导致费端还未收到 ZK 通知，提供者就停机了。也不能设置太长，太长又会导致关停应用时间边长，影响布体验。

## 3.3、注销 Protocol

```
ExtensionLoader<Protocol> loader = ExtensionLoader.getExtensionLoader(Protocol.class);
for (String protocolName : loader.getLoadedExtensions()) {
    try {
        Protocol protocol = loader.getLoadedExtension(protocolName);
        if (protocol != null) {
            protocol.destroy();
        }
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
```

```
    }
}
```

loader#getLoadedExtensions 将会返回两种 Protocol 子类，分别为 DubboProtocol 与 InjvmProtocol。

DubboProtocol 用与服务端请求交互，而 InjvmProtocol 用于内部请求交互。如果应用调用自己提供 Dubbo 服务，不会再执行网络调用，直接执行内部方法。

这里我们主要来分析一下 DubboProtocol 内部逻辑。

DubboProtocol#destroy 源码：

```
public void destroy() {
    // 关闭 Server
    for (String key : new ArrayList<String>(serverMap.keySet())) {
        ExchangeServer server = serverMap.remove(key);
        if (server != null) {
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Close dubbo server: " + server.getLocalAddress());
                }
                server.close(ConfigUtils.getServerShutdownTimeout());
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        }
    }
    // 关闭 Client
    for (String key : new ArrayList<String>(referenceClientMap.keySet())) {
        ExchangeClient client = referenceClientMap.remove(key);
        if (client != null) {
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Close dubbo connect: " + client.getLocalAddress() + "-->" + client.getRemoteAddress());
                }
                client.close(ConfigUtils.getServerShutdownTimeout());
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        }
    }

    for (String key : new ArrayList<String>(ghostClientMap.keySet())) {
        ExchangeClient client = ghostClientMap.remove(key);
        if (client != null) {
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Close dubbo connect: " + client.getLocalAddress() + "-->" + client.getRemoteAddress());
                }
                client.close(ConfigUtils.getServerShutdownTimeout());
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
```

```
            }
        }
    }
    stubServiceMethodsMap.clear();
    super.destroy();
}
```

Dubbo 默认使用 Netty 作为其底层的通讯框架，分为 Server 与 Client。Server 用于接收其他消费者 lient 发出的请求。

上面源码中首先关闭 Server ，停止接收新的请求，然后再关闭 Client。这样做就降低服务被消费者用的可能性。

## 3.4、关闭 Server

首先将会调用 HeaderExchangeServer#close，源码如下:

```
public void close(final int timeout) {
    startClose();
    if (timeout > 0) {
        final long max = (long) timeout;
        final long start = System.currentTimeMillis();
        if (getUrl().getParameter(Constants.CHANNEL_SEND_READONLYEVENT_KEY, true)) {
      // 发送 READ_ONLY 事件
            sendChannelReadOnlyEvent();
        }
        while (HeaderExchangeServer.this.isRunning()
                && System.currentTimeMillis() - start < max) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                logger.warn(e.getMessage(), e);
            }
        }
    }
    // 关闭定时心跳检测
    doClose();
    server.close(timeout);
}

private void doClose() {
    if (!closed.compareAndSet(false, true)) {
        return;
    }
    stopHeartbeatTimer();
    try {
        scheduled.shutdown();
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
    }
}
```

这里将会向服务消费者发送 READ_ONLY 事件。消费者接受之后，主动排除这个节点，将请求发往他正常节点。这样又进一步降低了注册中心通知延迟带来的影响。

接下来将会关闭心跳检测，关闭底层通讯框架 NettyServer。这里将会调用 NettyServer#close 方法这个方法实际在 AbstractServer 处实现。

AbstractServer#close 源码如下:

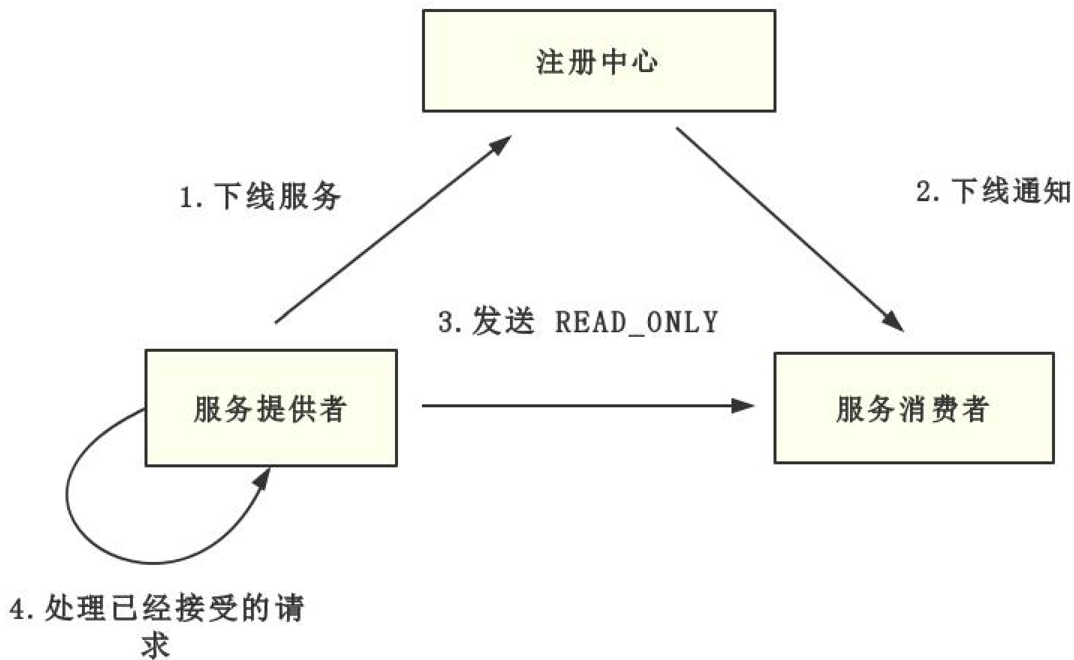
```
public void close(int timeout) {
    ExecutorUtil.gracefulShutdown(executor, timeout);
    close();
}
```

这里首先关闭业务线程池，这个过程将会尽可能将线程池中的任务执行完毕，再关闭线程池，最后在关闭 Netty 通讯底层 Server。

Dubbo 默认将会把请求/心跳等请求派发到业务线程池中处理。

关闭 Server，优雅等待线程池关闭，解决了上面提到的第二个问题：若关闭服务提供者，已经接收到务请求，需要处理完毕才能下线服务。

Dubbo 服务提供者关闭流程如图：




ps:为了方便调试源码，附上 Server 关闭调用联。

```
DubboProtocol#destroy
    ->HeaderExchangeServer#close
        ->AbstractServer#close
            ->NettyServer#doClose
```

## 3.5 关闭 Client

Client 关闭方式大致同 Server，这里主要介绍一下处理已经发出请求逻辑，代码位于HeaderExchangeChannel#close。

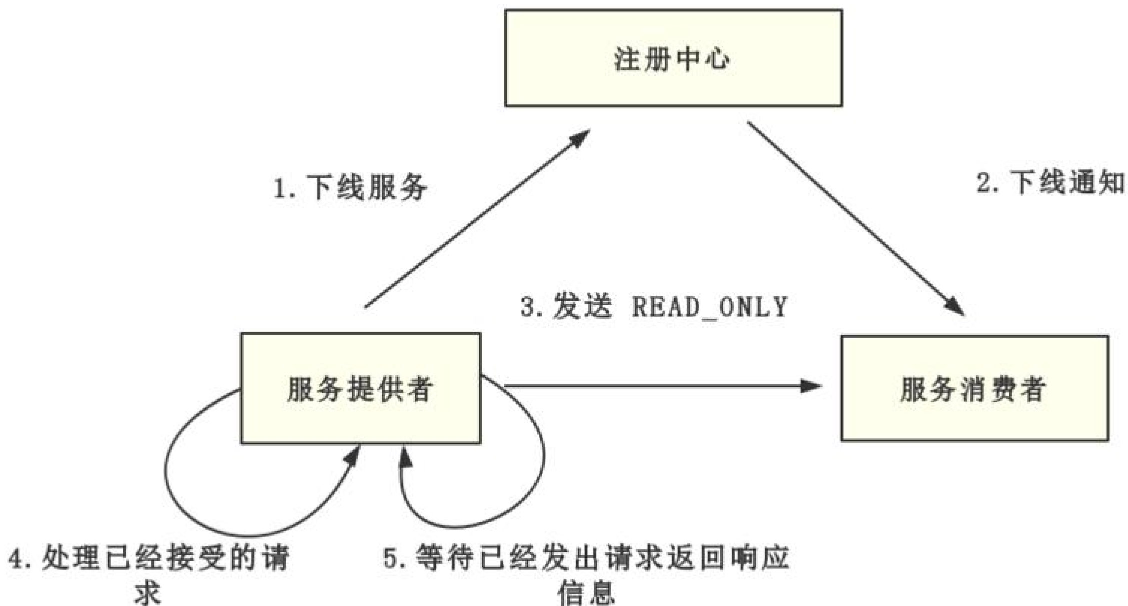
```
// graceful close
public void close(int timeout) {
    if (closed) {
        return;
    }
    closed = true;
    if (timeout > 0) {
        long start = System.currentTimeMillis();
    // 等待发送的请求响应信息
        while (DefaultFuture.hasFuture(channel)
                && System.currentTimeMillis() - start < timeout) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                logger.warn(e.getMessage(), e);
            }
        }
    }
    close();
}
```

关闭 Client 的时候，如果还存在未收到响应的信息请求，将会等待一定时间，直到确认所有请求都收响应，或者等待时间超过超时时间。

ps:Dubbo 请求会暂存在 DefaultFuture Map 中，所以只要简单判断一下 Map 就能知道请求是否都到响应。

通过这一点我们就解决了第三个问题：若关闭服务消费者，已经发出的服务请求，需要等待响应返回。

Dubbo 优雅停机总体流程如图所示。




ps: Client 关闭调用链如下所示：

```
DubboProtocol#close
    ->ReferenceCountExchangeClient#close
        ->HeaderExchangeChannel#close
            ->AbstractClient#close
```

# 2.7.X

Dubbo 一般与 Spring 框架一起使用，2.5.X 版本的停机过程可能导致优雅停机失效。这是因为 Sprin 框架关闭时也会触发相应的 ShutdownHook 事件，注销相关 Bean。这个过程若 Spring 率先执行机，注销相关 Bean。而这时 Dubbo 关闭事件中引用到 Spring 中 Bean，这就将会使停机过程中发异常，导致优雅停机失效。

为了解决该问题，Dubbo 在 2.6.X 版本开始重构这部分逻辑，并且不断迭代，直到 2.7.X 版本。

新版本新增 ShutdownHookListener，继承 Spring ApplicationListener 接口，用以监听 Spring 关事件。这里 ShutdownHookListener 仅仅监听 Spring 关闭事件，当 Spring 开始关闭，将会触发 hutdownHookListener 内部逻辑。

```
public class SpringExtensionFactory implements ExtensionFactory {
    private static final Logger logger = LoggerFactory.getLogger(SpringExtensionFactory.class);

    private static final Set<ApplicationContext> CONTEXTS = new ConcurrentHashSet<Applica
ionContext>();
    private static final ApplicationListener SHUTDOWN_HOOK_LISTENER = new ShutdownHoo
Listener();

    public static void addApplicationContext(ApplicationContext context) {
        CONTEXTS.add(context);
        if (context instanceof ConfigurableApplicationContext) {
            // 注册 ShutdownHook
            ((ConfigurableApplicationContext) context).registerShutdownHook();
            // 取消 AbstractConfig 注册的 ShutdownHook 事件
            DubboShutdownHook.getDubboShutdownHook().unregister();
        }
        BeanFactoryUtils.addApplicationListener(context, SHUTDOWN_HOOK_LISTENER);
    }
    // 继承 ApplicationListener，这个监听器将会监听容器关闭事件
    private static class ShutdownHookListener implements ApplicationListener {
        @Override
        public void onApplicationEvent(ApplicationEvent event) {
            if (event instanceof ContextClosedEvent) {
                DubboShutdownHook shutdownHook = DubboShutdownHook.getDubboShutdow
Hook();
                shutdownHook.doDestroy();
            }
        }
    }
}
```

当 Spring 框架开始初始化之后，将会触发 SpringExtensionFactory 逻辑，之后将会注销 AbstractC nfig 注册 ShutdownHook，然后增加 ShutdownHookListener。这样就完美解决上面『双 hook』问题。

# 最后

优雅停机看起来实现不难，但是里面设计细枝末节却非常多，一个点实现有问题，就会导致优雅停机
效。如果你也正在实现优雅停机，不妨参考一下 Dubbo 的实现逻辑。

# 帮助文章

1、强烈推荐阅读 kirito 大神文章：一文聊透 Dubbo 优雅停机

原文链接：Dubbo 优雅停机演进之路